LLNL-CONF-442476

Verification for Portability, Scalability, and Grokkability

A. Humphrey, C. Derrick, B. Tibbits, A. Vo, S. Vakkalanka, G. Gopalakrishnan, B. de Supinski, M. Schulz, G. Bronevetsky

July 15, 2010

EC2 Workshop
Edinburgh, United Kingdom
July 15, 2010 through July 19, 2010

Verification for Portability, Scalability, and Grokkability

Alan Humphrey[1], Christopher Derrick[1], Beth Tibbitts[2],
Anh Vo[1], Sarvani Vakkalanka[1], Ganesh Gopalakrishnan[1],
Bronis R. Supinski[3], Martin Schulz[3] and Greg Bronevetsky[3]

[1] School of Computing, University of Utah, UT, USA
[2]IBM Corp., Coldstream Research Campus, Lexington, KY
[3]Lawrence Livermore National Laboratory, Livermore, CA

The Message Passing Interface (MPI) API is widely used in almost all high performance computing applications. Yet, conventional debugging tools for MPI suffer from two serious drawbacks: they cannot prevent the exponentially growing number of redundant schedules from being explored; and they cannot prevent the processes from being locked into a small subset of schedules, unfortunately often reaching the potentially buggy schedules only when programs are ported to new platforms.

Dynamic verification methods are the natural choice for debugging real-world MPI programs when model extraction and maintenance are expensive. While many dynamic verification tools exist for verifying shared memory programs, there have been no corresponding tools that support MPI – the *lingua franca* of parallel programming.

Verification for Portability

While interleaving reduction suggests the use of dynamic partial order reduction (DPOR), four aspects of MPI make previous DPOR algorithms inapplicable: (i) MPI contains asynchronous calls that can complete out of program order; (ii) MPI has global synchronization operations that have weak semantics; (iii) the runtime of MPI cannot, without intrusive modifications, be forced to pursue a specific interleaving with non-deterministic wildcard receives; and (iv) the progress of MPI operations can depend on platform dependent runtime buffering, making bugs sometimes appear when resources are added to boost performance.

We will describe a formal model for MPI, and introduces a tailor-made notion of *Happens-Before* ordering for MPI functions. The crucial feature of this *Happens-Before* relation is that it elegantly solves all these four problems. MPI dynamic analysis is turned into a prioritized scheduling algorithm respecting MPI's *Happens-Before*.

We will describe three algorithms that have been demonstrated in the context of a practical MPI dynamic verification tool called ISP. The Partial Order avoiding Elusive Interleavings (POE) algorithm is a simple prioritized execution of the MPI transitions and is guaranteed to find all deadlocks, assertion violations and resource leaks under zero buffering. POE$_{OPT}$ algorithm avoids many of the redundant interleavings of POE by fully exploiting MPI's happens-before. Finally, the POE$_{MSE}$ algorithm discovers all possible minimal run-time bufferings that guarantee to discover bugs.

POE$_{MSE}$'s slack analysis has minimal overheads, and offers the power of verifying for safe portability by considering all relevant bufferings that might exist in various platforms.

In effect a program is dynamically verified not just with respect to the platform on which the tool is run, but also with respect to all platforms.

Verification for Scalability

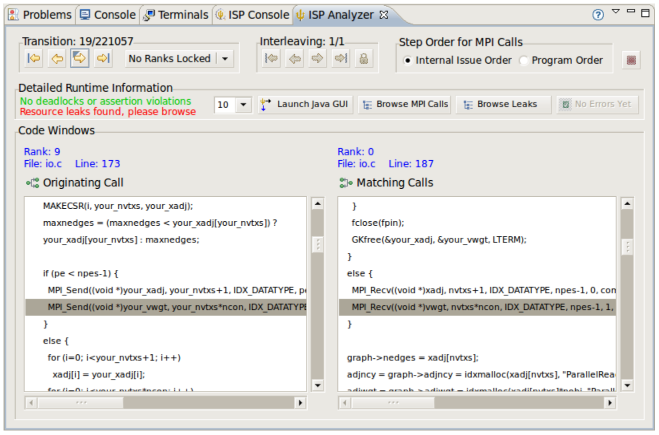
In order to make dynamic formal analysis tools practical, one must first of all scale up their capabilities to handle realistically sized examples. These examples have enormous CPU and memory needs that can be provided only by large clusters. In many cases it is impossible to (manually or automatically) downscale an MPI program that has been designed for large problem sizes. Such programs are often not well parameterized by having a few parameters that can be down/up scaled in a predictable way. Often the parameters themselves are not known. Often the relationships between these parameters is not known. Even if these are known, there are bugs that tend to show up (with respect to compilers, runtimes, etc.) only at scale.

Large applications are arrived at after a progression of testing and growing smaller prototypes. Each step up in problem-size requires optimizations. When the enhanced application finally breaks, it is very difficult to roll-back to a corresponding but downsized instance.

We achieve verification at scale by designing a new tool called Distributed MPI Analyzer (DMA). DMA employs a new distributed algorithm for scheduling MPI formal analysis. It employs piggyback messages that help track the causalities among the actual MPI messages, and determines *potential* matches between MPI's non-deterministic operations. DMA allows MPI programs to be run virtually at full speed, and on real cluster machines of upto 1000 cores, thus obtaining the CPU and memory scalability needed to run large MPI applications. The versatile instrumentation framework in DMA allows Fortran and C applications to be seamlessly verified – another important requirement of MPI formal analysis. DMA has been used to dynamically verify large MPI applications in a fraction of time that ISP takes.

Verification for Grokkability

It is important that designers be able to drive formal verification tools in the same context in which they launch jobs on clusters and conduct performance simulations. Such ability to operate seamlessly will minimize mistakes and also



Transition: 19/221057

Step Order for MPI Calls: [✓ Internal Issue Order] Program Order

Detailed Runtime Information

No deadlocks or assertion violations
Resource leaks found, please browse

10

Rank: 9
File: io.c Line: 173

```
    MAKECSR(i, your_nvtxs, your_xadj);
    maxnedges = (maxnedges < your_xadj[your_nvtxs]) ?
    your_xadj[your_nvtxs] : maxnedges;

    if (pe < npes-1) {
      MPI_Send((void *)your_xadj, your_nvtxs+1, IDX_DATATYPE, pe
      MPI_Send((void *)your_vwgt, your_nvtxs*ncon, IDX_DATATYPE
    }
    else {
      for (i=0; i<your_nvtxs+1; i++)
        xadj[i] = your_xadj[i];
```

Rank: 0
File: io.c Line: 187

```
   }
   fclose(fpin);
   GKfree(&your_xadj, &your_vwgt, LTERM);
  }
  else {
   MPI_Recv((void *)xadj, nvtxs+1, IDX_DATATYPE, npes-1, 0, con
   MPI_Recv((void *)vwgt, nvtxs*ncon, IDX_DATATYPE, npes-1, 1,
  }

  graph->nedges = xadj[nvtxs];
  adjncy = graph->adjncy = idxmalloc(xadj[nvtxs], "ParallelRea
```

Figure 1. Analyzer View on ParMETIS

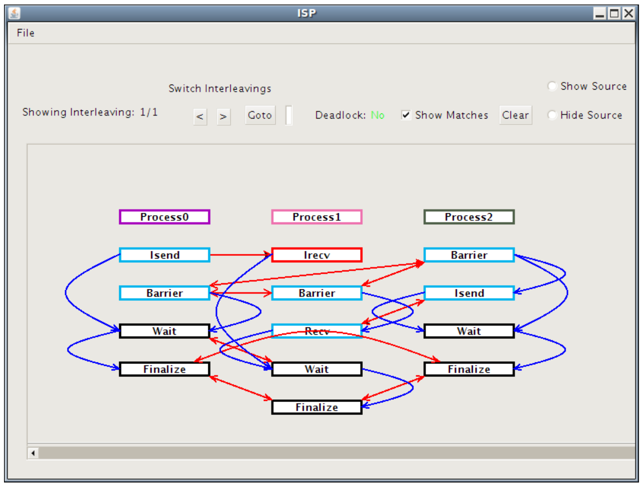


Figure 2. Happens-Before Viewer

encourage the use of formal analysis technologies. The integration has to occur in popular frameworks already being developed.

Our recent work is on such an official release of ISP (and soon DMA) as a plug-in architecture into the Eclipse Parallel Tools Platform Version 3.0 released in December 2009.

The talk

Our talk will emphasize all these aspects of formal verification. Details are at `http://www.cs.utah.edu/fv`

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.